**TypeScript Theory:**

**What is TypeScript?**

TypeScript is an open-source object-oriented language developed and maintained by Microsoft.

It is a typed superset of JavaScript that compiles to plain JavaScript.

TypeScript extends JavaScript by adding data types, classes, and other object-oriented features with type-checking.

**Why TypeScript?**

JavaScript is a dynamic programming language with no type system. JavaScript provides primitive types like string, number, object, etc., but it doesn't check assigned values. JavaScript variables are declared using the var keyword, and it can point to any value. JavaScript doesn't support classes and other object-oriented features (ECMA2015 supports it). So, without the type system, it is not easy to use JavaScript to build complex applications with large teams working on the same code.

The type system increases the code quality, readability and makes it easy to maintain and refactor codebase. More importantly, errors can be caught at compile time rather than at runtime.

Hence, the reason to use TypeScript is that it catches errors at compile-time, so that you can fix it before you run code. It supports object-oriented programming features like data types, classes, enums, etc., allowing JavaScript to be used at scale.

TypeScript compiles into simple JavaScript. The TypeScript compiler is also implemented in TypeScript and can be used with any browser.

**Benefits of TypeScript:**

1) TypeScript is an open-source language with continuous development and maintenance by Microsoft.

2) TypeScript runs on any browser or JavaScript engine.

3) TypeScript is similar to JavaScript and uses the same syntax and semantics. All of TypeScript's code finally gets converted into JavaScript. This allows a quicker learning curve for front-end developers currently coding in JavaScript.

4) TypeScript is also closer in syntax to backend languages like Java and Scala. This helps backend developers write front-end code faster.

5) TypeScript code can be called from an existing JavaScript code. TypeScript also works with existing JavaScript frameworks and libraries without any issues.

6) The TypeScript Definition file, with .d.ts extension, provides support for existing JavaScript libraries like jQuery, D3.js, etc. So, TypeScript code can add JavaScript libraries using type definitions to avail the benefits of type-checking, code autocompletion, and documentation in existing dynamically-typed JavaScript libraries.

**Installing TypeScript:**

**Install TypeScript using NPM:**

NPM (Node.js package manager) is used to install the TypeScript package on your local machine or a project. Make sure you have Node.js install on your local machine. If you are using JavaScript frameworks for your application, then it is highly recommended to install Node.js.

To install or update the latest version of TypeScript, open command prompt/terminal and type the following command:

**npm install -g typescript**

The above command will install TypeScript globally so that you can use it in any project. Check the installed version of TypeScript using the following command:

**tsc -v**

To know few more ways to install the TypeScript, refer to the following URL: [Install TypeScript (tutorialsteacher.com)](https://www.tutorialsteacher.com/typescript/typescript-environment-setup)

**TypeScript Application Structure:**

**Reference URL:** [How does TypeScript work? The bird’s eye view • Tackling TypeScript (exploringjs.com)](https://exploringjs.com/tackling-ts/ch_typescript-workflows.html)

This is one possible file structure for TypeScript projects:

typescript-project/

dist/

ts/

src/

main.ts

util.ts

test/

util\_test.ts

tsconfig.json

Explanations:

* Directory ts/ contains the TypeScript files:
  + Subdirectory ts/src/ contains the actual code.
  + Subdirectory ts/test/ contains tests for the code.
* Directory dist/ is where the output of the compiler is stored.
* The TypeScript compiler compiles TypeScript files in ts/ to JavaScript files in dist/. For example:
  + ts/src/main.ts is compiled to dist/src/main.js (and possibly other files)
* tsconfig.json is used to configure the TypeScript compiler.

**Importance of tsconfig.json File:**

The contents of tsconfig.json look as follows:

{

"compilerOptions": {

"rootDir": "ts",

"outDir": "dist",

"module": "commonjs",

···

}

}

We have specified that:

* The root directory of the TypeScript code is ts/.
* The directory where the TypeScript compiler saves its output is dist/.
* The module format of the output files is CommonJS.

## **TypeScript Playground**

TypeScript provides an online playground <https://www.typescriptlang.org/play> to write and test your code on the fly without the need to download or install anything.

This is a great place for beginners to learn TypeScript and try different TypeScript features.

**TypeScript Types:**

1) **Number:** Just like JavaScript, TypeScript supports number data type. All numbers are stored as floating-point numbers.

**E.g. -**

let first:number = 123;

let second:number = 10.89;

2) **String:** String is another primitive data type that is used to store text data. String values are surrounded by single quotation marks or double quotation marks.

**E.g. -**

let employeeName:string = 'John Smith';

//OR

let employeeName:string = "John Smith";

3) **Boolean:** Boolean values are supported by both JavaScript and TypeScript and stored as true/false values.

**E.g. –**

let isPresent:boolean = true;

Note that, the boolean Boolean is different from the lower case boolean type. The upper case Boolean is an object type whereas lower case boolean is a primitive type. It is recommended to use the primitive type boolean in your code, because, while JavaScript coerces an object to its primitive type, the TypeScript type system does not. TypeScript treats it like an object type.

So, instead of using upper case function checkExistence(b: Boolean), use the lower case function checkExistence(b: boolean) boolean type.

**Type Inference in TypeScript:**

TypeScript is a typed language. However, it is not mandatory to specify the type of a variable. TypeScript infers types of variables when there is no explicit information available in the form of type annotations.

Types are inferred by TypeScript compiler when:

Variables are initialized

Default values are set for parameters

Function return types are determined

For example,

var a = "some text";

Here, since we are not explicitly defining a: string with a type annotation, TypeScript infers the type of the variable based on the value assigned to the variable.

The value of a is a string and hence the type of a is inferred as string.

Consider the following example:

var a = "some text";

var b = 123;

a = b; // Compiler Error: Type 'number' is not assignable to type 'string'

The above code shows an error because while inferring types, TypeScript inferred the type of variable a as string and variable b as number.

When we try to assign b to a, the compiler complains saying a number type cannot be assigned to a string type.

**Array:**

An array is a special type of data type which can store multiple values of different data types sequentially using a special syntax.

TypeScript supports arrays, similar to JavaScript.

There are two ways to declare an array:

1. Using square brackets. This method is similar to how you would declare arrays in JavaScript.

let fruits: string[] = ['Apple', 'Orange', 'Banana'];

2. Using a generic array type, Array<elementType>.

let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];

Both methods produce the same output.

Of course, you can always initialize an array like shown below, but you will not get the advantage of TypeScript's type system.

let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];

Arrays can contain elements of any data type, numbers, strings, or even objects.

Arrays can be declared and initialized separately.

Example: Array Declaration and Initialization

let fruits: Array<string>;

fruits = ['Apple', 'Orange', 'Banana'];

let ids: Array<number>;

ids = [23, 34, 100, 124, 44];

An array in TypeScript can contain elements of different data types using a generic array type syntax, as shown below.

Example: Multi Type Array

let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

// or

let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];

**Accessing Array Elements:**

The array elements can be accessed using the index of an element e.g. ArrayName[index]. The array index starts from zero, so the index of the first element is zero, the index of the second element is one and so on.

Example: Access Array Elements

let fruits: string[] = ['Apple', 'Orange', 'Banana'];

fruits[0]; // returns Apple

fruits[1]; // returns Orange

fruits[2]; // returns Banana

fruits[3]; // returns undefined

**Use the for loop to access array elements as shown below.**

Example: Access Array Elements using Loop Copy

let fruits: string[] = ['Apple', 'Orange', 'Banana'];

for(var index in fruits)

{

console.log(fruits[index]); // output: Apple Orange Banana

}

for(var i = 0; i < fruits.length; i++)

{

console.log(fruits[i]); // output: Apple Orange Banana

}

**Enums:**

Enums or enumerations are a new data type supported in TypeScript. Most object-oriented languages like Java and C# use enums. This is now available in TypeScript too.

In simple words, enums allow us to declare a set of named constants i.e. a collection of related values.

Enums can be defined using the keyword enum. Let's say we want to store a set of print media types. The corresponding enum in TypeScript would be:

**E.g. -**

enum PrintMedia {

Newspaper,

Newsletter,

Magazine,

Book

}

In the above example, we have an enum named PrintMedia. The enum has four values: Newspaper, Newsletter, Magazine, and Book. Here, enum values start from zero and increment by 1 for each member. It would be represented as:

Newspaper = 0

Newsletter = 1

Magazine = 2

Book = 3

Enums are always assigned numeric values when they are stored. The first value always takes the numeric value of 0, while the other values in the enum are incremented by 1.

We also have the option to initialize the first numeric value ourselves. For example, we can write the same enum as:

enum PrintMedia {

Newspaper = 1,

Newsletter,

Magazine,

Book

}

The first member, Newspaper, is initialized with the numeric value 1. The remaining members will be incremented by 1 from the numeric value of the first value. Thus, in the above example, Newsletter would be 2, Magazine would be 3 and Book would be 4.

It is not necessary to assign sequential values to Enum members. They can have any values.

enum PrintMedia {

Newspaper = 1,

Newsletter = 5,

Magazine = 5,

Book = 10

}

**Union:**

TypeScript allows us to use more than one data type for a variable or a function parameter. This is called union type.

**E.g. –**

let code: (string | number);

code = 123; // OK

code = "ABC"; // OK

code = false; // Compiler Error

let empId: string | number;

empId = 111; // OK

empId = "E111"; // OK

empId = true; // Compiler Error

In the above example, variable code is of union type, denoted using (string | number). So, you can assign a string or a number to it.

**Any:**

TypeScript has type-checking and compile-time checks. However, we do not always have prior knowledge about the type of some variables, especially when there are user-entered values from third party libraries. In such cases, we need a provision that can deal with dynamic content. The Any type comes in handy here.

**E.g. –**

let something: any = "Hello World!";

something = 23;

somethi The above code will compile into the following JavaScript.

var something = "Hello World!";

something = 23;

something = true;

Similarly, you can create an **array of type any[]** if you are not sure about the types of values that can contain this array.

**E.g. -**

let arr: any[] = ["John", 212, true];

arr.push("Smith");

console.log(arr); //Output: [ 'John', 212, true, 'Smith' ]

The above example will generate the following JavaScript code:

var arr = ["John", 212, true];

arr.push("Smith");

console.log(arr);ng = true;

ES 6 Features in TypeScript:

TypeScript - Variable

TypeScript follows the same rules as JavaScript for variable declarations. Variables can be declared using: var, let, and const.

var

Variables in TypeScript can be declared using var keyword, same as in JavaScript. The scoping rules remains the same as in JavaScript.

let

To solve problems with var declarations, ES6 introduced two new types of variable declarations in JavaScript, using the keywords let and const. TypeScript, being a superset of JavaScript, also supports these new types of variable declarations.

Example: Variable Declaration using let:

let employeeName = "John";

// or

let employeeName:string = "John";

The let declarations follow the same syntax as var declarations. Unlike variables declared with var, variables declared with let have a block-scope. This means that the scope of let variables is limited to their containing block, e.g. function, if else block or loop block. Consider the following example.

Example: let Variables Scope

let num1:number = 1;

function letDeclaration() {

let num2:number = 2;

if (num2 > num1) {

let num3: number = 3;

num3++;

}

while(num1 < num2) {

let num4: number = 4;

num1++;

}

console.log(num1); //OK

console.log(num2); //OK

console.log(num3); //Compiler Error: Cannot find name 'num3'

console.log(num4); //Compiler Error: Cannot find name 'num4'

}

let Declaration();

In the above example, all the variables are declared using let. num3 is declared in the if block so its scope is limited to the if block and cannot be accessed out of the if block. In the same way, num4 is declared in the while block so it cannot be accessed out of while block. Thus, when accessing num3 and num4 else where will give a compiler error.

The same example with the var declaration is compiled without an error.

Example: var Variables Scope

var num1:number = 1;

function varDeclaration() {

var num2:number = 2;

if (num2 > num1) {

var num3: number = 3;

num3++;

}

while(num1 < num2) {

var num4: number = 4;

num1++;

}

console.log(num1); //2

console.log(num2); //2

console.log(num3); //4

console.log(num4); //4

}

varDeclaration();

**Const**

Variables can be declared using const similar to var or let declarations. The const makes a variable a constant where its value cannot be changed. Const variables have the same scoping rules as let variables.

**E.g. - :**

**Const Variable:**

const num:number = 100;

num = 200; //Compiler Error: Cannot assign to 'num' because it is a constant or read-only property

Const variables must be declared and initialized in a single statement. Separate declaration and initialization is not supported.

const num:number; //Compiler Error: const declaration must be initialized

num = 100;

Const variables allow an object sub-property to be changed but not the object structure.

**E.g. –**

**Const Object:**

const playerCodes = {

player1 : 9,

player2 : 10,

player3 : 13,

player4 : 20

};

playerCodes.player2 = 11; // OK

playerCodes = { //Compiler Error: Cannot assign to playerCodes because it is a constant or read-only

player1 : 50, // Modified value

player2 : 10,

player3 : 13,

player4 : 20

};

Even if you try to change the object structure, the compiler will point this error out.

const playerCodes = {

player1: 9,

player2: 10,

player3: 13,

player4: 20

};

playerCodes = { //Compiler Error: Cannot assign to playerCodes because it is a constant or read-only

player1: 9,

player2: 10,

player3: 13,

player4: 20,

player5: 22

};

**Arrow Functions:**

Fat arrow notations are used for anonymous functions i.e. for function expressions. They are also called lambda functions in other languages.

**Syntax:**

(param1, param2, ..., paramN) => expression

Using fat arrow =>, we dropped the need to use the function keyword. Parameters are passed in the parenthesis (), and the function expression is enclosed within the curly brackets { }.

**E.g. –**

let sum = (x: number, y: number): number => {

return x + y;

}

sum(10, 20); //returns 30

In the above example, sum is an arrow function. (x:number, y:number) denotes the parameter types, :number specifies the return type. The fat arrow => separates the function parameters and the function body. The right side of => can contain one or more code statements.

The above arrow function sum will be converted into the following JavaScript code.

var sum = function (x, y) {

return x + y;

}

The following is an **arrow function without parameters**.

let Print = () => console.log("Hello TypeScript");

Print(); //Output: Hello TypeScript

Furthermore, if the function body consists of only one statement then no need for the curly brackets and the return keyword, as shown below.

let sum = (x: number, y: number) => x + y;

sum(3, 4); //returns 7

**A class can include an arrow function as a property, as shown below.**

class Employee {

empCode: number;

empName: string;

constructor(code: number, name: string) {

this.empName = name;

this.empCode = code;

}

display = () => console.log(this.empCode +' ' + this.empName)

}

let emp = new Employee(1, 'Ram');

emp.display();

**Default Function Parameters:**

**Reference URL:** [TypeScript Default Parameters (typescripttutorial.net)](https://www.typescripttutorial.net/typescript-tutorial/typescript-default-parameters/)

If you don’t pass arguments or pass the undefined into the function when calling it, the function will take the default initialized values for the omitted parameters.

E.g. –

function applyDiscount(price: number, discount: number = 0.05): number {

return price \* (1 - discount);

}

console.log(applyDiscount(100)); //Output: 95

**Spread and Rest Operators:**

**Reference URL:** [Rest and Spread parameters in TypeScript. | by alfred a. mohenu | Medium](https://medium.com/@alfmohenu/rest-and-spread-parameters-in-typescript-e98db065d518)

When the number of parameters that a function will receive is not known or can vary, we can use rest parameters.

We can pass zero or more arguments to the rest parameter. The compiler will create an array of arguments with the rest parameter name provided by us.

**E.g. –**

function Greet(greeting: string, ...names: string[]) {

return greeting + " " + names.join(", ") + "!";

}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"

In the above example, we have a function with two parameters: greeting and names. Here, names is a rest parameter denoted by ellipses .... While calling the function, we first pass "Steve", "Bill" as the rest parameters. These are combined into a string array by joining the elements of the names array. Hence, it returns "Hello Steve, Bill!". During the second function call, we do not pass any arguments as the rest parameters. This is accepted by the compiler and hence the output is "Hello !"

The rest parameters can be used in functions, arrow functions or classes.

let Greet = (greeting: string, ...names: string[]) => {

return greeting + " " + names.join(", ") + "!";

}

Greet("Hello", "Steve", "Bill"); // returns "Hello Steve, Bill!"

Greet("Hello");// returns "Hello !"

**Spread**

Rest is to gather whiles Spread is to scatter. Say we have an array with 3 values and we have a function that accepts 3 arguments, we can scatter or spread the values in the array into the function.

const arr = [1,2,3];

function Test(a,b,c) {

console.log(a); // 1

console.log(b); // 2

console.log(c); // 3

}

Test(...arr); // spreading the array into the function

**Destructuring:**

Destructuring assignment is a syntax that allows you to assign object properties or array items as variables.

This can greatly reduce the lines of code necessary to manipulate data in these structures.

There are two types of destructuring: Object destructuring and Array destructuring.

**Object Destructuring:**

Object destructuring allows you to create new variables using an object property as the value.

**E.g. -**

const note = {

id: 1,

title: 'My first note',

date: '01/01/1970',

}

const { id, title, date } = note;

console.log(id);

console.log(title);

console.log(date);

**Array Destructuring:**

Array destructuring allows you to create new variables using an array item as a value.

const doj = ['2005', '12', '01'];

const[year, month, day] = doj;

console.log(year);

console.log(month);

console.log(day);

const[yoj,,dateOfJoining] = doj;

console.log(yoj);

console.log(dateOfJoining);

**Template String Literals:**

**Reference URL:** [TypeScript: Documentation - Template Literal Types (typescriptlang.org)](https://www.typescriptlang.org/docs/handbook/2/template-literal-types.html)

Template String Literals have ability to expand into strings.

**E.g. –**

let World = "world";

let Greeting = `hello ${World}`;

//Output: let Greeting = Hello World

**OOP in TypeScript:**

**Classes:**

In object-oriented programming languages like Java and C#, classes are the fundamental entities used to create reusable components. Functionalities are passed down to classes and objects are created from classes. However, until ECMAScript 6 (also known as ECMAScript 2015), this was not the case with JavaScript. JavaScript has been primarily a functional programming language where inheritance is prototype-based. Functions are used to build reusable components. In ECMAScript 6, object-oriented class based approach was introduced. TypeScript introduced classes to avail the benefit of object-oriented techniques like encapsulation and abstraction. The class in TypeScript is compiled to plain JavaScript functions by the TypeScript compiler to work across platforms and browsers.

A class can include the following:

Constructor

Properties

Methods

The following is an example of a class in TypeScript:

class Employee {

empCode: number;

empName: string;

constructor(code: number, name: string) {

this.empName = name;

this.empCode = code;

}

getSalary() : number {

return 10000;

}

}

The TypeScript compiler will convert the above class to the following JavaScript code using closure:

var Employee = /\*\* @class \*/ (function () {

function Employee(name, code) {

this.empName = name;

this.empCode = code;

}

Employee.prototype.getSalary = function () {

return 10000;

};

return Employee;

}());

**Constructor**

The constructor is a special type of method which is called when creating an object. In TypeScript, the constructor method is always defined with the name "constructor".

Example: Constructor

class Employee {

empCode: number;

empName: string;

constructor(empcode: number, name: string ) {

this.empCode = empcode;

this.name = name;

}

}

In the above example, the Employee class includes a constructor with the parameters empcode and name. In the constructor, members of the class can be accessed using this keyword e.g. this.empCode or this.name.

**Access Modifiers:**

In object-oriented programming, the concept of 'Encapsulation' is used to make class members public or private i.e. a class can control the visibility of its data members. This is done using access modifiers.

There are three types of access modifiers in TypeScript: public, private and protected.

**public**

By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

Example: public

class Employee {

public empCode: string;

empName: string;

}

let emp = new Employee();

emp.empCode = 123;

emp.empName = "Swati";

In the above example, empCode and empName are declared as public. So, they can be accessible outside of the class using an object of the class.

Please notice that there is not any modifier applied before empName, as TypeScript treats properties and methods as public by default if no modifier is applied to them.

**private**

The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

Example: private

class Employee {

private empCode: number;

empName: string;

}

let emp = new Employee();

emp.empCode = 123; // Compiler Error

emp.empName = "Swati";//OK

In the above example, we have marked the member empCode as private. Hence, when we create an object emp and try to access the emp.empCode member, it will give an error.

**protected**

The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

Example: protected

class Employee {

public empName: string;

protected empCode: number;

constructor(name: string, code: number){

this.empName = name;

this.empCode = code;

}

}

class SalesEmployee extends Employee{

private department: string;

constructor(name: string, code: number, department: string) {

super(name, code);

this.department = department;

}

}

let emp = new SalesEmployee("John Smith", 123, "Sales");

empObj.empCode; //Compiler Error

In the above example, we have a class Employee with two members, public empName and protected property empCode. We create a subclass SalesEmployee that extends from the parent class Employee. If we try to access the protected member from outside the class, as emp.empCode, we get the following compilation error:

error TS2445: Property 'empCode' is protected and only accessible within class 'Employee' and its subclasses.

**ReadOnly:**

TypeScript includes the readonly keyword that makes a property as read-only in the class.

Prefix readonly is used to make a property as read-only. Read-only members can be accessed outside the class, but their value cannot be changed. Since read-only members cannot be changed outside the class, they either need to be initialized at declaration or initialized inside the class constructor.

Example: ReadOnly Class Properties

class Employee {

readonly empCode: number;

empName: string;

constructor(code: number, name: string) {

this.empCode = code;

this.empName = name;

}

}

let emp = new Employee(10, "John");

emp.empCode = 20; //Compiler Error

emp.empName = 'Bill';

In the above example, we have the Employee class with two properties- empName and empCode. Since empCode is read only, it can be initialized at the time of declaration or in the constructor.

If we try to change the value of empCode after the object has been initialized, the compiler shows the following compilation error:

error TS2540: Cannot assign to empCode' because it is a constant or a read-only property.

**Static:**

ES6 includes static members and so does TypeScript. The static members of a class are accessed using the class name and dot notation, without creating an object e.g. <ClassName>.<StaticMember>.

The static members can be defined by using the keyword static.

Consider the following example of a class with static members.

class Circle {

static pi: number = 3.14;

static calculateArea(radius:number) {

return this.pi \* radius \* radius;

}

}

Circle.pi; // returns 3.14

Circle.calculateArea(5); // returns 78.5

The above Circle class includes a static property and a static method. Inside the static method calculateArea, the static property can be accessed using this keyword or using the class name Circle.pi.

**Properties: Getters and Setters:**

**Reference URL:** [TypeScript Getter & Setters (typescripttutorial.net)](https://www.typescripttutorial.net/typescript-tutorial/typescript-getters-setters/)

The getters and setters allow you to control the access to the properties of a class.

For each property:

A getter method returns the value of the property’s value. A getter is also called an accessor.

A setter method updates the property’s value. A setter is also known as a mutator.

A getter method starts with the keyword get and a setter method starts with the keyword set.

**E.g. –**

class Person {

private \_age: number;

private \_firstName: string;

private \_lastName: string;

public get age() {

return this.\_age;

}

public set age(theAge: number) {

if (theAge <= 0 || theAge >= 200) {

throw new Error('The age is invalid');

}

this.\_age = theAge;

}

public getFullName(): string {

return `${this.\_firstName} ${this.\_lastName}`;

}

}

**Inheritance using extends Keyword:**

Just like object-oriented languages such as Java and C#, TypeScript classes can be extended to create new classes with inheritance, using the keyword extends.

Example: Inheritance

class Person {

name: string;

constructor(name: string) {

this.name = name;

}

}

class Employee extends Person {

empCode: number;

constructor(empcode: number, name:string) {

super(name);

this.empCode = empcode;

}

displayName():void {

console.log("Name = " + this.name + ", Employee Code = " + this.empCode);

}

}

let emp = new Employee(100, "Bill");

emp.displayName(); // Name = Bill, Employee Code = 100

In the above example, the Employee class extends the Person class using extends keyword. This means that the Employee class now includes all the members of the Person class.

The constructor of the Employee class initializes its own members as well as the parent class's properties using a special keyword 'super'. The super keyword is used to call the parent constructor and passes the property values.

**Note:**

We must call super() method first before assigning values to properties in the constructor of the derived class.

**Abstract Classes:**

Define an abstract class in Typescript using the abstract keyword. Abstract classes are mainly for inheritance where other classes may derive from them. We cannot create an instance of an abstract class.

An abstract class typically includes one or more abstract methods or property declarations. The class which extends the abstract class must define all the abstract methods.

The following abstract class declares one abstract method find and also includes a normal method display.

**Example: Abstract Class**

abstract class Person {

name: string;

constructor(name: string) {

this.name = name;

}

display(): void{

console.log(this.name);

}

abstract find(string): Person;

}

class Employee extends Person {

empCode: number;

constructor(name: string, code: number) {

super(name); // must call super()

this.empCode = code;

}

display(): void{

console.log(this.empCode+” “+this.name);

}

find(name:string): Person {

// execute AJAX request to find an employee from a db

return new Employee(name, 1);

}

}

let emp: Person = new Employee("James", 100);

emp.display(); //James

let emp2: Person = emp.find('Steve');

In the above example, Person is an abstract class which includes one property and two methods, one of which is declared as abstract. The find() method is an abstract method and so must be defined in the derived class. The Employee class derives from the Person class and so it must define the find() method as abstract. The Employee class must implement all the abstract methods of the Person class, otherwise the compiler will show an error.

**Note:**

The class which implements an abstract class must call super() in the constructor.

**Interfaces:**

Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

The TypeScript compiler does not convert interface to JavaScript. It uses interface for type checking. This is also known as "duck typing" or "structural subtyping".

An interface is defined with the keyword interface and it can include properties and method declarations using a function or an arrow function.

**Example: Interface**

interface IEmployee

empCode: number;

empName: string;

getSalary: (number) => number; // arrow function

getManagerName(number): string;

}

In the above example, the IEmployee interface includes two properties empCode and empName. It also includes a method declaration getSalaray using an arrow function which includes one number parameter and a number return type. The getManagerName method is declared using a normal function. This means that any object of type IEmployee must define the two properties and two methods.

**Implementing an Interface**

Similar to languages like Java and C#, interfaces in TypeScript can be implemented with a Class. The Class implementing the interface needs to strictly conform to the structure of the interface.

**Example: Interface Implementation**

interface IEmployee {

empCode: number;

name: string;

getSalary:(empCode: number) => number;

}

class Employee implements IEmployee {

empCode: number;

name: string;

constructor(code: number, name: string) {

this.empCode = code;

this.name = name;

}

getSalary(empCode:number):number {

return 20000;

}

}

let emp = new Employee(1, "Steve");

In the above example, the IEmployee interface is implemented in the Employee class using the the implement keyword. The implementing class should strictly define the properties and the function with the same name and data type. If the implementing class does not follow the structure, then the compiler will show an error.

Of course, the implementing class can define extra properties and methods, but at least it must define all the members of an interface.

**Interface as Function Type**

TypeScript interface is also used to define a type of a function. This ensures the function signature.

**Example: Function Type**

interface KeyValueProcessor

{

(key: number, value: string): void;

};

function addKeyValue(key:number, value:string):void {

console.log('addKeyValue: key = ' + key + ', value = ' + value)

}

function updateKeyValue(key: number, value:string):void {

console.log('updateKeyValue: key = '+ key + ', value = ' + value)

}

let kvp: KeyValueProcessor = addKeyValue;

kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill

kvp = updateKeyValue;

kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve

In the above example, an interface KeyValueProcessor includes a method signature. This defines the function type. Now, we can define a variable of type KeyValueProcessor which can only point to functions with the same signature as defined in the KeyValueProcessor interface. So, addKeyValue or updateKeyValue function is assigned to kvp. So, kvp can be called like a function.

**Trying to assign a function with a different signature will cause an error.**

function delete(key:number):void {

console.log('Key deleted.')

}

let kvp: KeyValueProcessor = delete; //Compiler Error

**Method or Function Overloading:**

TypeScript provides the concept of function overloading. You can have multiple functions with the same name but different parameter types and return type. However, the number of parameters should be the same.

**Example: Function Overloading**

function add(a:string, b:string):string;

function add(a:number, b:number): number;

function add(a: any, b:any): any {

return a + b;

}

add("Hello ", "Steve"); // returns "Hello Steve"

add(10, 20); // returns 30

In the above example, we have the same function add() with two function declarations and one function implementation. The first signature has two parameters of type string, whereas the second signature has two parameters of the type number.

The last function should have the function implementation. Since the return type can be either string or number as per the first two function declarations, we must use compatible parameters and return type as any in the function definition.

**Generics:**

When writing programs, one of the most important aspects is to build reusable components. This ensures that the program is flexible as well as scalable in the long-term.

Generics offer a way to create reusable components. Generics provide a way to make components work with any data type and not restrict to one data type. So, components can be called or used with a variety of data types. Generics in TypeScript is almost similar to C# generics.

Let's see why we need Generics using the following example.

function getArray(items : any[] ) : any[] {

return new Array().concat(items);

}

let myNumArr = getArray([100, 200, 300]);

let myStrArr = getArray(["Hello", "World"]);

myNumArr.push(400); // OK

myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // OK

myStrArr.push(500); // OK

console.log(myNumArr); // [100, 200, 300, 400, "Hi"]

console.log(myStrArr); // ["Hello", "World", "Hello TypeScript", 500]

In the above example, the getArray() function accepts an array of type any. It creates a new array of type any, concats items to it and returns the new array. Since we have used type any for our arguments, we can pass any type of array to the function. However, this may not be the desired behavior. We may want to add the numbers to number array or the strings to the string array but not numbers to the string array or vice-versa.

To solve this, TypeScript introduced generics. Generics uses the type variable <T>, a special kind of variable that denotes types. The type variable remembers the type that the user provides and works with that particular type only. This is called preserving the type information.

The above function can be rewritten as a generic function as below.

**Example: Generic Function**

function getArray<T>(items : T[] ) : T[] {

return new Array<T>().concat(items);

}

let myNumArr = getArray<number>([100, 200, 300]);

let myStrArr = getArray<string>(["Hello", "World"]);

myNumArr.push(400); // OK

myStrArr.push("Hello TypeScript"); // OK

myNumArr.push("Hi"); // Compiler Error

myStrArr.push(500); // Compiler Error

In the above example, the type variable T is specified with the function in the angle brackets getArray<T>. The type variable T is also used to specify the type of the arguments and the return value. This means that the data type which will be specified at the time of a function call, will also be the data type of the arguments and of the return value.

We call generic function getArray() and pass the numbers array and the strings array. For example, calling the function as getArray<number>([100, 200, 300]) will replace T with the number and so, the type of the arguments and the return value will be number array. In the same way, for getArray<string>(["Hello", "World"]), the type of arguments and the return value will be string array. So now, the compiler will show an error if you try to add a string in myNumArr or a number in myStrArr array. Thus, you get the type checking advantage.

It is not recommended but we can also call a generic function without specifying the type variable. The compiler will use type inference to set the value of T on the function based on the data type of argument values.

Example: Calling Generic Function without Specifying the Type Copy

let myNumArr = getArray([100, 200, 300]); // OK

let myStrArr = getArray(["Hello", "World"]); // OK

Generics can be applied to the function's argument, a function's return type, and a class fields or methods.

**TypeScript - Generic Classes**

TypeScript supports generic classes. The generic type parameter is specified in angle brackets after the name of the class. A generic class can have generic fields (member variables) or methods.

**Example: Generic Class**

class KeyValuePair<T,U>

{

private key: T;

private val: U;

setKeyValue(key: T, val: U): void {

this.key = key;

this.val = val;

}

display():void {

console.log(`Key = ${this.key}, val = ${this.val}`);

}

}

let kvp1 = new KeyValuePair<number, string>();

kvp1.setKeyValue(1, "Steve");

kvp1.display(); //Output: Key = 1, Val = Steve

let kvp2 = new KayValuePair<string, string>();

kvp2.SetKeyValue("CEO", "Bill");

kvp2.display(); //Output: Key = CEO, Val = Bill

In the above example, we created a generic class named KeyValuePair with a type variable in the angle brackets <T, U>. The KeyValuePair class includes two private generic member variables and a generic function setKeyValue that takes two input arguments of type T and U. This allows us to create an object of KeyValuePair with any type of key and value.

**Generic Constraints:**

**Reference URL:** [TypeScript Generic Constraints (typescripttutorial.net)](https://www.typescripttutorial.net/typescript-tutorial/typescript-generic-constraints/)

Consider the following example:

function merge<U, V>(obj1: U, obj2: V) {

return {

...obj1,

...obj2

};

}

Code language: TypeScript (typescript)

The merge() is a generic function that merges two objects.

For example:

let person = merge(

{ name: 'John' },

{ age: 25 }

);

console.log(result);

Code language: TypeScript (typescript)

Output:

{ name: 'John', age: 25 }

Code language: TypeScript (typescript)

It works perfectly fine.

The merge() function expects two objects. However, it doesn’t prevent you from passing a non-object like this:

let person = merge(

{ name: 'John' },

25

);

console.log(person);

Code language: TypeScript (typescript)

Output:

{ name: 'John' }

Code language: TypeScript (typescript)

TypeScript doesn’t issue any error.

Instead of working with all types, you may want to add a constraint to the merge() function so that it works with objects only.

To do this, you need to list out the requirement as a constraint on what U and V types can be.

In order to denote the constraint, you use the extends keyword. For example:

function merge<U extends object, V extends object>(obj1: U, obj2: V) {

return {

...obj1,

...obj2

};

}

Code language: TypeScript (typescript)

Because the merge() function is now constrained, it will no longer work with all types. Instead, it works with the object type only.

The following will result in an error:

let person = merge(

{ name: 'John' },

25

);

Code language: TypeScript (typescript)

Error:

Argument of type '25' is not assignable to parameter of type 'object'.

**Decorators in TypeScript:**

**Reference URL:** [TypeScript: Documentation - Decorators (typescriptlang.org)](https://www.typescriptlang.org/docs/handbook/decorators.html)

**Why Decorators?**

With the introduction of Classes in TypeScript and ES6, there now exist certain scenarios that require additional features to support annotating or modifying classes and class members. Decorators provide a way to add both annotations and a meta-programming syntax for class declarations and members.

To enable experimental support for decorators, you must enable the experimentalDecorators compiler option either on the command line or in your tsconfig.json:

Command Line:

tsc --target ES5 --experimentalDecorators

tsconfig.json:

{

"compilerOptions": {

"target": "ES5",

"experimentalDecorators": true

}

}

**What are Decorators?**

A Decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators use the form @expression, where expression must evaluate to a function that will be called at runtime with information about the decorated declaration.

For example, given the decorator @sealed we might write the sealed function as follows:

function sealed(target) {

// do something with 'target' ...

}

**Modules in TypeScript:**

TypeScript provides modules and namespaces to organize and maintain a large code base.

Modules are a way to create a local scope in the file. So, all variables, classes, functions, etc. that are declared in a module are not accessible outside the module. A module can be created using the keyword export and a module can be used in another module using the keyword import.

**Export**

A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes. Use the prefix export with all the definitions you want to include in a module and want to access from other modules.

E.g. -

Employee.ts

export let age : number = 20;

export class Employee {

empCode: number;

empName: string;

constructor(name: string, code: number) {

this.empName = name;

this.empCode = code;

}

displayEmployee() {

console.log ("Employee Code: " + this.empCode + ", Employee Name: " + this.empName );

}

}

let companyName:string = "XYZ";

In the above example, Employee.ts is a module which contains two variables and a class definition. The age variable and the Employee class are prefixed with the export keyword, whereas companyName variable is not. Thus, Employee.ts is a module which exports the age variable and the Employee class to be used in other modules by importing the Employee module using the import keyword. The companyName variable cannot be accessed outside this Employee module, as it is not exported.

**Import**

A module can be used in another module using an import statement.

**Syntax:**

Import { export name } from "file path without extension"

Let's see different ways of importing a module export.

**Importing a Single export from a Module:**

We exported a variable and a class in the Employee.ts. However, we can only import the export module which we are going to use. The following code only imports the Employee class from Employee.ts into another module in the EmployeeProcessor.ts file.

EmployeeProcessor.ts

import { Employee } from "./Employee";

let empObj = new Employee("Steve Jobs", 1);

empObj.displayEmployee(); //Output: Employee Code: 1, Employee Name: Steve Jobs

Importing the Entire Module into a Variable

You can import all the exports in a module as shown below.

EmployeeProcessor.ts

import \* as Emp from "./Employee"

console.log(Emp.age); // 20

let empObj = new Emp.Employee("Bill Gates" , 2);

empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: Bill Gates

In the above example, we import all the exports in Employee module in a single variable called Emp. So, we don't need to write an export statement for each individual module. In the above example, it will import age and Employee class into the Emp variable and can be accessed using Emp.age and Emp.Employee.